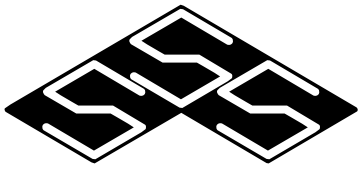
Applications of Ptolemy in Securities Trading

or,
Playing the Markets with Ptolemy

Tom Lane

Structured Software Systems, Inc.



Playing the Markets with Ptolemy

- We are developing systems for automated securities trading.

- We follow a technical, not fundamental, trading philosophy.

  We look at patterns of price activity to decide when to buy/sell, rather than studying underlying data such as economic reports, corporate P&L statements, etc. (Eventually we will add sources of fundamental data to our systems.)

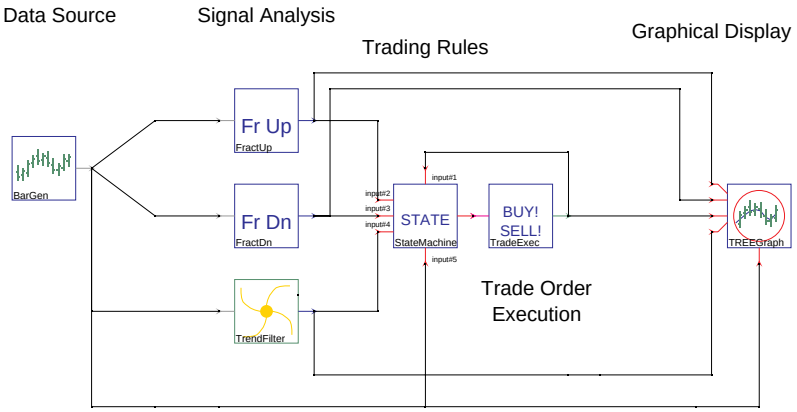
- Ptolemy is useful because it provides a solid framework for combining multiple design notations and execution semantics.

  We use C++ code for signal processing, Tcl scripts for trading rules, and Vem schematics to plug the pieces together.

A simple trading system

- **Domain is a variant of DE**
  - **Timestamps represent real time**
- **Specialized data types:**
  - **Price bars**
  - **Trade orders**
  - **Trade status**
- **Also good old float & int**

Williams' "Fractal" Breakout Trading System



- **Bar generator stars produce price bars by reading historical database and/or accessing real-time datafeed**
- **Trade executor stars model execution of a trade order to estimate actual execution price (for historical testing) or send order to operator and receive execution price (for live trading)**
- **Specialized graphical display star**

Self-contained Tcl scripts

We avoid needing a C++ star definition for each trading script by picking up port and parameter definitions on-the-fly from the script. For example, the Williams fractal script includes:

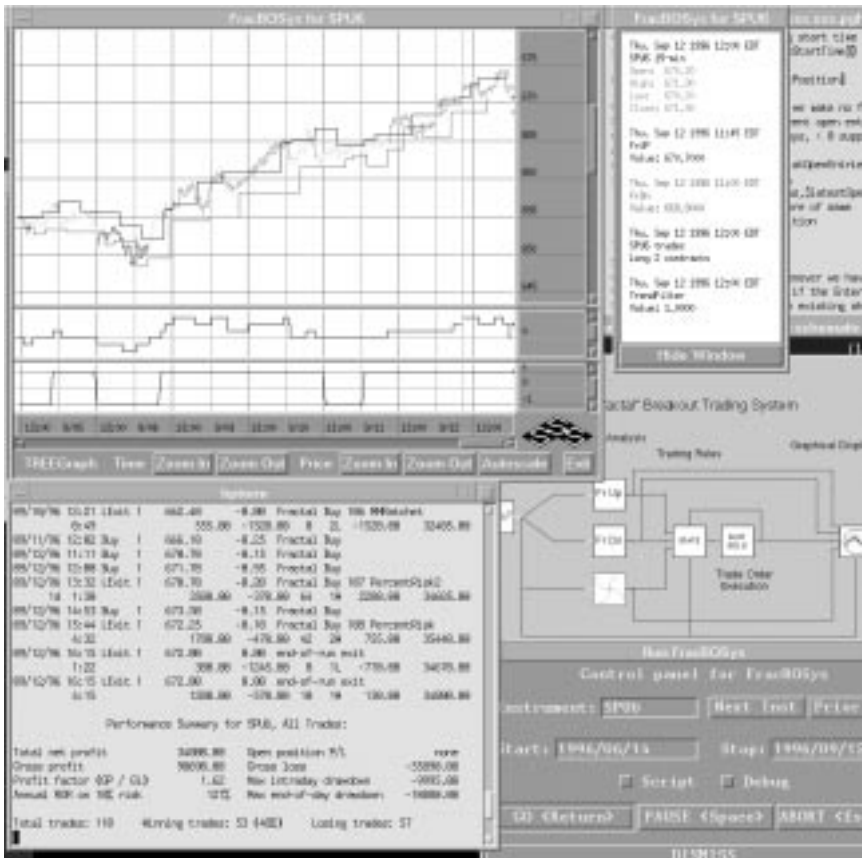
```
# Inputs are assumed to be set up as follows:
define-input TStatus1 tstatus      "Trade status report"
define-input FractUp2 float        "Up-fractal detection report"
define-input FractDn3 float        "Down-fractal detection report"
define-input Filter 4 float        "Additional filter permitting entry"
define-input Bars   5 bar          "Bar series for instrument"

# Output ports are:
define-output Output1 torder       "Output trade orders"

# Star parameters are:
define-parameter TicksPenetration int 1 "Enter N ticks past fractal peak"
define-parameter CheckLeverage int TRUE "Whether to use loss-of-leverage check"
define-parameter MaxContracts int 5     "Max # contracts to hold"
define-parameter UseFractStop int TRUE  "Whether to enable fractal stop rule"
define-parameter StopTicksPenetration int 1 "Ticks past peak for fractal stop"
# the script uses standard stop rule packages that define additional parameters
```

Historical testing example

- Modified control panel sets start/stop dates and target instrument
- Trade log & summary statistics can be written to a file (here, log appears in shell window)
- Graphical display facilities:
  - Scroll and zoom for easier examination of long data series
  - Multiple configurable panes for display of data series of dissimilar ranges
  - Adequate performance with long data series
  - Popup cursor window allows exact values to be read off
  - Incremental update while run proceeds

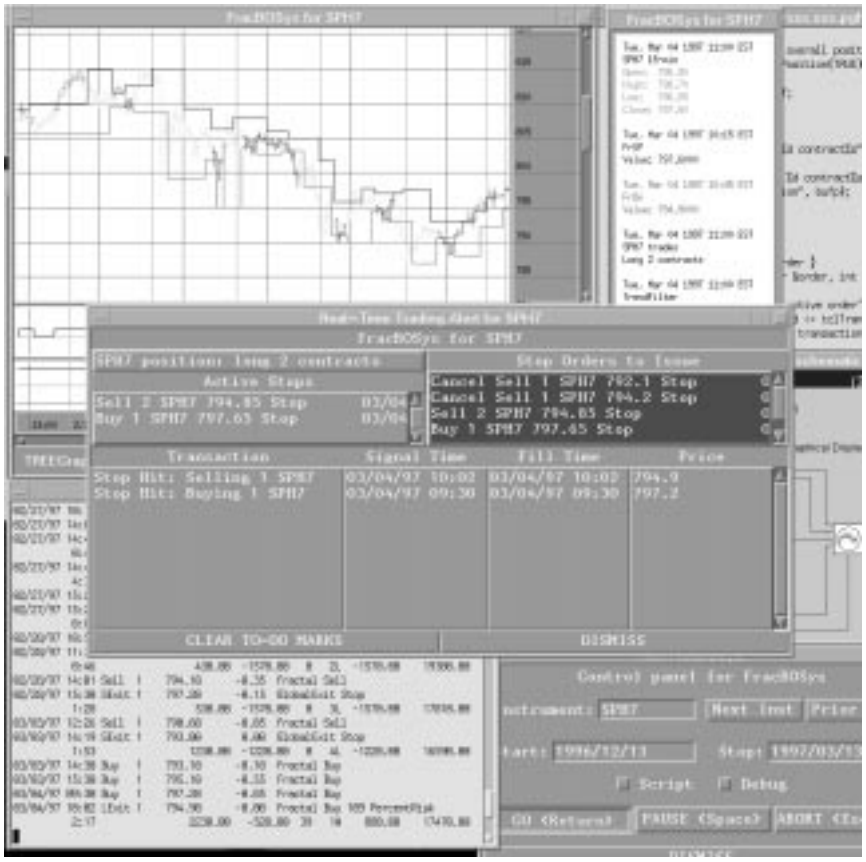


Live trading

- We rarely start a trading system “from scratch” at current real time, because signal processors such as moving averages need some historical data to converge to correct values. So a trading system is started on historical data old enough to let it reach steady state.
- When simulated time advances as far as current real time, bar generators and trade executors automatically begin to fetch tick data from the live datafeed instead of the historical database.
- Trade executors can be configured to report orders to operator via popup window, and accept entry of actual execution price (overriding executor’s own estimate that it makes by watching the ticks go by).
- Scheduler is also aware of real time, and does not allow simulated time to advance beyond current real time. The system sleeps between arrivals of new ticks from the datafeed; CPU usage decreases drastically compared to historical simulation.
- No other part of the system has any clue whether trading is live or not.

Live trading example

- Screen dump shows a run that was started at simulated date 12/13/96, and "went live" as of the morning of 3/4/97.
- Two transactions were already completed today. The system is currently long (has bought) two S&P futures contracts.
- It is 11:00AM, and the trading rules have just decided to move the sell stop levels up a few points, as well as buy another contract if the price rises one more tick. The popup window notifies the operator to place these orders.



Sub-universes

- We often need to examine the same interval of time repeatedly; for example, to test a trading system with different parameter combinations. Since time cannot go backwards within a universe, this means repeated runs of a universe.
- Ptolemy 0.6 supports repeated runs with ptcl scripts (in ptcl, or in pigi run control window) but that's not flexible enough; we want to apply analysis schematics to our results.
- We make an entire run of a "sub-universe" be a single event in an outer controlling universe. The sub-universe is controlled by a Tcl-scripted star. Each such star has a private PTcl object to hold its sub-universes.
- We added some features to allow the control script to extract particle values from the sub-universe (it can retrieve the last particle sent to any "Register" star in the sub-universe). It can examine the particle values and optionally re-emit them into the outer universe. In particular, an entire trade history (emitted by a TradeExec star) can be recovered thus.
- Schematics can be run standalone or as subuniverses without change.

Causality is critical

- Obviously, a realtime trading system cannot "look into the future"; it can only know prices up to the current instant.
- In traditional historical testing setups, it's not hard to violate this condition by accident. For example, an off-by-one array subscript might use today's closing price in an expression that was supposed to use yesterday's close. The resulting system is likely to do spectacularly well at predicting today's prices ... but only in historical testing.
- Conceptual errors are also possible — for example, we know a widely used commercial charting package that labels bars with their opening times, thus effectively assuming that the closing price of a bar is available at the bar's opening time.
- In our Ptolemy systems, if we trust the scheduler not to run time backwards, and we trust BarGen and TradeExec not to read ticks ahead of the current simulated time, we know we have no causality violations.
- This is very comforting.

But what about causality with subuniverses?

- The subuniverse feature re-opens Pandora's box: it's easy to make causality-related mistakes.
- Wrong: run a trading system 100 times over same interval, take best parameter set, claim that we could actually have gotten that result in real time. (We couldn't have known which set was best until too late.)
- Right: optimize parameter set by testing over a given interval, then use best parameter values for actual trading in next interval. Repeat for each interval. This technique is called "walk-forward optimization", and it works nicely if best parameters change slowly over time.
- The difference between valid and invalid procedure is mighty narrow, and it can be hard to see at the level of scripting. How can we protect ourselves against foolish errors?

Solution: “real” vs. “hypothetical” subuniverse runs

- A run of a subuniverse is labeled hypothetical if it might have access to noncausal information. If only non-hypothetical runs are considered to represent achievable trading results, we protect ourselves against causality errors.
- These are the rules:
  1. Any output event from a sub-universe-controlling star is timestamped with the later of the current time (in outer domain) and the latest sub-run end time the star has ever requested (in previous firings as well as this one). This prevents data from being emitted to the outer simulation sooner than it could be known.
  2. If the control star requests a sub-run that starts before the current time in the outer domain or the latest end time of any prior sub-run, then that sub-run is flagged as hypothetical (a flag which is automatically attached to all trade histories generated within it). This indicates that the run may be tainted by use of information not truly available at its start time.

“Real” vs. “hypothetical” subuniverse runs

- Example: to do walk-forward optimization, the control star fires once for each optimization time interval. It first performs look-back optimization runs ending at the current time, then a “real” run starting at current time and ending at the next interval boundary. Results of the look-back runs will be marked hypothetical, but the “real” run’s results will not be. Notice that you get only one chance for a “real” run per interval, which accords with how the real world works. Also note that the “real” run’s results are not available to the rest of the outer simulation until the end of the interval it covers.
- Rule 2 is conservative, and may flag runs that don’t actually depend on any tainted data. Doing better seems to require detailed analysis of data flow. These rules have sufficed for our purposes.
- Limitations: Inner and outer domain must use same definition of time, since we compare their timestamps. “Back door” communication paths out of control star are not plugged (eg, writing a file rather than emitting a Ptolemy event).

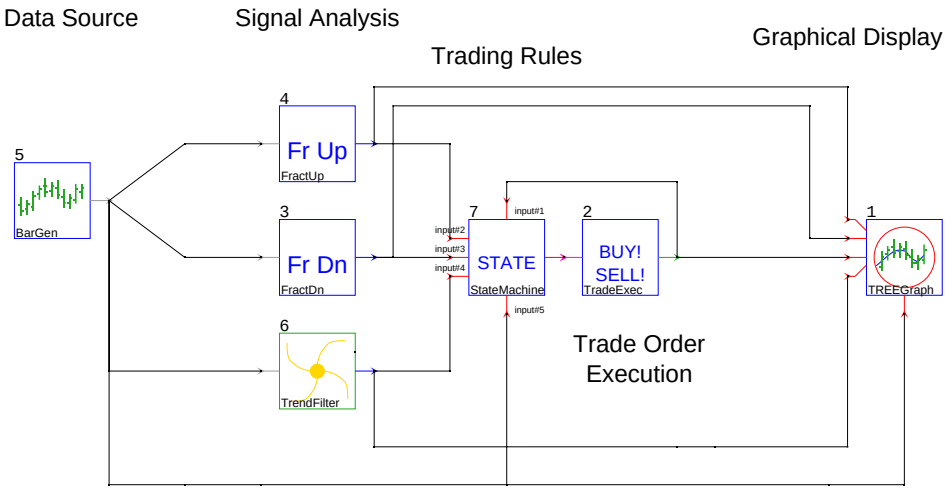
Where we'd like to see Ptolemy evolve in the future

- **We hate Vem ... and Oct too. We'd sure like to see a more modern schematic editor and a less inflexible design database.**
- **Tcl-scripted stars should be first-class citizens — one should not have to prepare a C-level star definition to support each new script. We have done some of the work needed to eliminate a C-level definition; not all.**
- **Tcl scripts are too slow. Tcl 8.0 might help a little, but probably another language is the only real answer. Auto conversion of SDF to CGC, followed by use of CreateSDFStar target, is an example of what we need: with those, you can develop in a quick-turnaround environment and then reduce the design to a faster-running block.**
- **We'd like more support for re-using designs in multiple execution contexts, and more support for developing customized execution contexts. Run-control-panel scripts are a good first step, but they don't offer much leverage. We want to do things like develop and test a schematic standalone, then plug it into some larger structure *without hand modification*. Automated alteration of schematics seems essential, but ptcl script language isn't quite able to do it.**

FracBOSys

Location: /users/frank/tree/system/FracBOSys Domain: TS Target: default-TS

Williams' "Fractal" Breakout Trading System

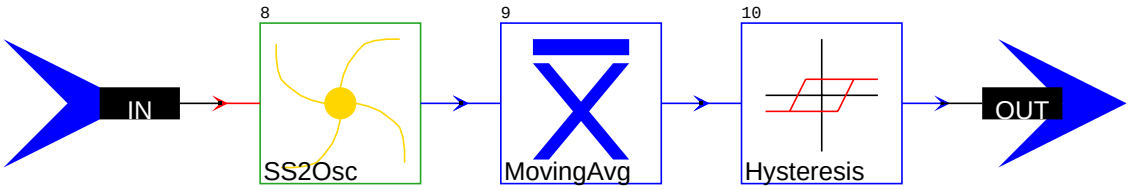


03/05/97 16:24 Structured Software Systems Page 1/6

TrendFilter

Location: /users/frank/tree/osc/TrendFilter Domain: TS Target: <parent>

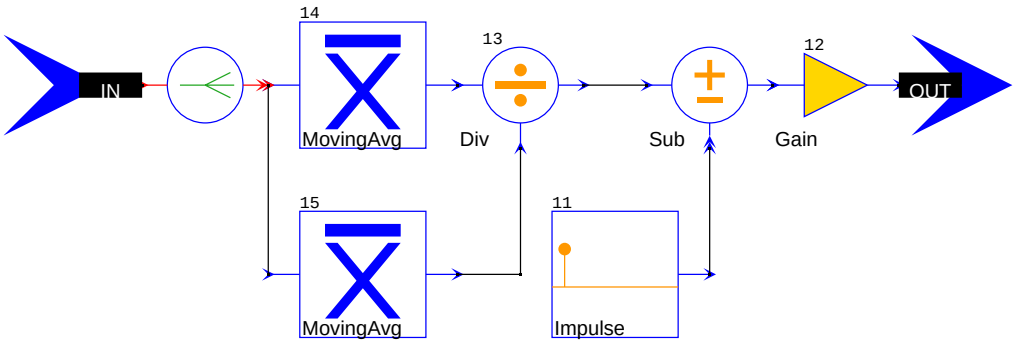
TrendFilter



SS2Osc

Location: /users/frank/play/SS2Osc Domain: TS Target: <parent>

SS2 Osc



FracBOSys Parameters

Domain: TS Target: default-TS

```
Top-level parameters
    STRING schematicID=FracBOSys
    STRING instrument=SPU6
    INT FracN=2
    INT standalone=TRUE

1  TREEGraph.input=41 parameters
    STRING title="{schematicID} for {instrument}"
    STRINGARRAY streamLabel=bars FrUP FrDn
      trades TrendFilter
    STRING configFile=TRX1.xtg
    INT waitTime=2
    INT enabled=standalone

2  TradeExec1 parameters
    STRING instrumentName="{instrument}"
    FLOAT fillDelay=1
    FLOAT stopFillDelay=".15"
    INT exitAtRunEnd=TRUE
    INT usePullback=TRUE
    FLOAT slipTicks=2
    STRING mktHours=DAY
    STRING startInteractive=""
    STRING tradeLogFile="<stdout>"
    STRING summaryFile="<stdout>"
    INT longSummary=FALSE
    STRING logTitle="{schematicID} for {
      instrument}"
    INT enableFiles=standalone

3  FractDn1 parameters
    INT N=FracN

4  FractUp1 parameters
    INT N=FracN

5  BarGen1 parameters
    STRING instrumentName="{instrument}"
    INT barWidth=15
    STRING barType=MIN
    INT barsBack=100
    STRING mktHours=DAY

6  TrendFilter1 parameters
    INT fast=3
    INT slow=10
    INT ma=16
    INT hys=10
    STRING Function=SAVGABS
    INT HysPeriod=34
```



```
7  StateMachine.input=5.output=11 parameters
    STRING tcl_file=FracBOSys.tcl
    INT debug_level=0
    STRING debug_log=""
    STRING instrument="{instrument}"
    DYNAMIC tcl_file=""
    INT TicksPenetration=1
    INT CheckLeverage=TRUE
    INT MaxContracts=3
    INT UseFractStop=TRUE
    INT StopTicksPenetration=1
    STRING ExitEODStates=none
    FLOAT ProfitTargetAmt=1000
    STRING ProfitTargetStates=none
    FLOAT MMRatchetStop=1500
    STRING MMRatchetRef=None
    STRING MMRatchetStates=""
    FLOAT PercentRiskFloor=1500
    FLOAT PercentRiskPercent=50
    STRING PercentRiskStates=""
    FLOAT PercentRisk2Floor=3000
    FLOAT PercentRisk2Percent=75
    STRING PercentRisk2States=""
```

TrendFilter1 Parameters

Domain: TS

```
8  SS2Osc1 parameters
    INT Fast=3
    INT Slow=10
    STRING AvgType=Simple

9  MovingAvg1 parameters
    STRING type=Simple
    FLOAT period=ma

10  Hysteresis1 parameters
    STRING function="{Function}"
    INT period=HysPeriod
    FLOAT factor=hys
    FLOAT bias=0
    FLOAT sdGain=1.0
    INT noMemory=FALSE
```

SS2Osc1 Parameters

Domain: TS

```
11  Impulse1 parameters
    FLOAT magnitude=1.0

12  Gain1 parameters
    FLOAT gain=100

13  Div1 parameters
    STRING onZeroDivide=AbortRun

14  MovingAvg1 parameters
    STRING type="{AvgType}"
    FLOAT period=Fast

15  MovingAvg2 parameters
    STRING type="{AvgType}"
    FLOAT period=Slow
```



```
# StateMachine script to implement Williams-style fractal breakout system.

# On detection of an up fractal, we set an entry buy stop N ticks above the
# fractal peak.  Similarly, detection of a down fractal sets an entry sell
# stop N ticks below the fractal peak.
# An open entry stop is canceled if the price exceeds the prior opposite-
# direction fractal peak (Williams calls this loss of leverage).

# An entry will be taken only when the "filter" input does not disagree:
# a filter input above 0 disables short entries, below 0 disables long entries.
# (Stop exits will still be taken, however.  Note that the filter does not
# force us out of an existing position, only prevent getting further in.)

# Multiple entries in the same direction can result when successive fractal
# breakouts occur in the same direction.  But a breakout in the opposite
# direction causes us to reverse (close out all entries and go to one contract
# long or short).

# Optionally, stops can be set to get out of an entry before reversal.
# In addition to standard stop exit rules, we provide a stop at the worse
# of the last 2 opposite-direction fractal peaks (Williams' "airbag" exit).

# Inputs are assumed to be set up as follows:
define-input TStatus     1 tstatus         "Trade status report"
define-input FractUp     2 float           "Up-fractal detection report"
define-input FractDn     3 float           "Down-fractal detection report"
define-input Filter      4 float           "Additional filter permitting
entry"
define-input Bars        5 bar             "Bar series for instrument"

# Output ports are:
define-output Output     1 torder          "Output trade orders"

# Star parameters are:
define-parameter TicksPenetration int 1 "Enter N ticks past fractal peak"
define-parameter CheckLeverage int TRUE "Whether to use loss-of-leverage check"
define-parameter MaxContracts int 5 "Max # contracts to hold"
define-parameter UseFractStop int TRUE  "Whether to enable fractal stop rule"
define-parameter StopTicksPenetration int 1 "Ticks past peak for fractal stop"

# Convert tick-scaled parameters to price moves
set OneTick [minMove $instrument]
set Penetration [expr $OneTick * $TicksPenetration]
set StopPenetration [expr $OneTick * $StopTicksPenetration]

# We use an order handler
NewOrderHandler myOH $TStatus $Output
myOH SetMaxPosition $MaxContracts

# We allow one of each kind of standard stop.
source ExitEOD.tcl
source ProfitTarget.tcl
source MMRatchet.tcl
source PercentRisk.tcl
source PercentRisk2.tcl

### Script state variables ###
# The up-fractal and down-fractal recognizers are each in one of two states:
#   NotReady    no fractal seen yet, or canceled by loss of leverage
#   Ready       a buy or sell stop should be set
set UpFractState NotReady
set DnFractState NotReady

# We keep track of last five fractal high and low points
data-buffer UpFractHighs 5 $FractUp
data-buffer DnFractLows 5 $FractDn

### State definitions ###

# We don't have much use for a global state in this script,
# so we just use one state.
initial-state MainState

define-state MainState {} {
    # Check for fractal state transitions
    if {newInput($FractUp) && [DnFractLows available] > 0} {
       set UpFractState Ready
       set UpFractEntry [expr input($FractUp) + $Penetration]
       set UpFractLeverage [DnFractLows get 0]
       set UpFractInitialStop [minVal [DnFractLows get 0 2]]
    }
    if {newInput($FractDn) && [UpFractHighs available] > 0} {
       set DnFractState Ready
       set DnFractEntry [expr input($FractDn) - $Penetration]
       set DnFractLeverage [UpFractHighs get 0]
       set DnFractInitialStop [maxVal [UpFractHighs get 0 2]]
    }
    # Check for cancellation of fractal signal by loss of leverage
    if {$CheckLeverage && newInput($Bars)} {
       if {$UpFractState == "Ready" && barLow($Bars) < $UpFractLeverage} {
           set UpFractState NotReady
       }
       if {$DnFractState == "Ready" && barHigh($Bars) > $DnFractLeverage} {
           set DnFractState NotReady
       }
    }
} {
```

```
    # Do nothing more until trading start time arrives
    if {[currentTime] < [simulationStartTime]} { return }

    set MarketPosition [myOH MarketPosition]

    # If we have an open position, we make no further entries of the
    # same type unless the most recent open entry is profitable.
    # EnterFilter > 0 suppresses buys, < 0 suppresses sells
    set EnterFilter 0
    set latestOpenEntry [lindex [listOpenEntries $TStatus] 0]
    if {$latestOpenEntry != {} && \
           entryOpenProfit($TStatus,$latestOpenEntry) <= 0} {
       # we're losing, suppress more of same
       set EnterFilter $MarketPosition
    }

    # Process buy signals
    # We want to set a buy stop whenever we have a fractal high to do it with.
    # A full buy is permitted only if the EnterFilter and Filter input allow;
    # if so, we go one long from an existing short position, or one more long
    # if already long.
    # If the filters do not allow a buy, we use the buy only as an exit stop
    # from an existing short position.
    if {$UpFractState == "Ready" && $MarketPosition < $MaxContracts} {
       if {$EnterFilter <= 0 && input($Filter) >= 0} {
           # full buy permitted
           myOH IssueEntryOrder "Fractal Buy" BuyStop 1 \
                $UpFractEntry hitBuyEntry
       } else {
           myOH CancelEntryOrder "Fractal Buy"
           if {$MarketPosition < 0} {
                # short exit permitted
                myOH SetGlobalExit Stop $UpFractEntry
           }
       }
    } else {
       myOH CancelEntryOrder "Fractal Buy"
    }

    # Process sell signals
    ... snipped for space --- just the inverse of the buy logic above ...
}


# Position-entry callbacks: need to disable signals
# to avoid re-issuing entry order

proc hitBuyEntry {orderID type numContracts entryPrice positionID entryIndex} {
    global UpFractState
    set UpFractState NotReady
}

proc hitSellEntry {orderID type numContracts entryPrice positionID entryIndex} {
    global DnFractState
    set DnFractState NotReady
}


# Stop manager for fractal stop rule

if {$UseFractStop} {

define-post-action UpdateFractalStop {
    foreach posID [myOH ListOpenPositions] {
       if {! [info exists curFractStop($posID)]} {
           # Need to compute the initial stop for this position
           if {[myOH GetPositionSize $posID] > 0} {
                set curFractStop($posID)  \
                        [expr $UpFractInitialStop-$StopPenetration]
           } else {
                set curFractStop($posID)  \
                        [expr $DnFractInitialStop+$StopPenetration]
           }
       }
       # Check latest two opposite-direction fractals,
       # move stop if more favorable
       if {[myOH GetPositionSize $posID] > 0} {
           # long: try to move up
           set secondFract [minVal [DnFractLows get 0 2]]
           set stop [expr $secondFract-$StopPenetration]
           if {$stop > $curFractStop($posID)} {
                set curFractStop($posID) $stop
           }
       } else {
           # short: try to move down
           set secondFract [maxVal [UpFractHighs get 0 2]]
           set stop [expr $secondFract+$StopPenetration]
           if {$stop < $curFractStop($posID)} {
                set curFractStop($posID) $stop
           }
       }
       myOH SetPositionExit $posID FractalStop Stop 0 $curFractStop($posID)
    }
}

} # matches outer if
```